Service vs provider vs factory

What are the differences between AngularJS module's `Service` , `Provider` and `Factory` ?

angularjs | dependency-injection | angularjs-service | angularjs-factory | angularjs-provider

edited Jun 23 at 13:34

Tim Castelijns
8,806 8 17 45

asked Mar 27 '13 at 17:59

Lior
12k 8 22 35

111 I found that all the Angular terms were intimidating for beginners. We started off with this cheatsheet that was a little easier for our programmers to understand while learning Angular demisx.github.io/angularjs/2014/09/14/…. Hope this helps your team too. – demisx Sep 16 '14 at 17:07

2 Here is the best explanation youtube.com/watch?v=J6qr6Wx3VPs – Shibin Ragh May 3 at 16:40

In my opinion, the best way to understand the difference is using Angular's own documentation: docs.angularjs.org/guide/providers it is extremely well explained and uses a peculiar example to help you understand it. – Rafael Merlin May 12 at 14:28

22 Answers

From the AngularJS mailing list I got an amazing thread that explains service vs factory vs provider and their injection usage. Compiling the answers:

Services

Syntax: `module.service( 'serviceName', function );`
Result: When declaring serviceName as an injectable argument **you will be provided with an instance of the function. In other words** `new FunctionYouPassedToService()` .

Factories

Syntax: `module.factory( 'factoryName', function );`
Result: When declaring factoryName as an injectable argument you will be provided with **the value that is returned by invoking the function reference passed to module.factory**.

Providers

Syntax: `module.provider( 'providerName', function );`
Result: When declaring providerName as an injectable argument **you will be provided with** `ProviderFunction().$get()` . The constructor function is instantiated before the $get method is called - ProviderFunction is the function reference passed to module.provider.

Providers have the advantage that they can be configured during the module configuration phase.

See here for the provided code.

Here's a great further explanation by Misko:

```
provide.value('a', 123);

function Controller(a) {
  expect(a).toEqual(123);
}
```

In this case the injector simply returns the value as is. But what if you want to compute the value? Then use a factory

```
provide.factory('b', function(a) {
  return a*2;
});

function Controller(b) {
  expect(b).toEqual(246);
}
```

So `factory` is a function which is responsible for creating the value. Notice that the factory function can ask for other dependencies.

But what if you want to be more OO and have a class called Greeter?

```
function Greeter(a) {
  this.greet = function() {
    return 'Hello ' + a;
  }
}
```

Then to instantiate you would have to write

```
provide.factory('greeter', function(a) {
  return new Greeter(a);
});
```

Then we could ask for 'greeter' in controller like this

```
function Controller(greeter) {
  expect(greeter instanceof Greeter).toBe(true);
  expect(greeter.greet()).toEqual('Hello 123');
}
```

But that is way too wordy. A shorter way to write this would be `provider.service('greeter', Greeter);`

But what if we wanted to configure the `Greeter` class before the injection? Then we could write

```
provide.provider('greeter2', function() {
  var salutation = 'Hello';
  this.setSalutation = function(s) {
    salutation = s;
  }

  function Greeter(a) {
    this.greet = function() {
      return salutation + ' ' + a;
    }
  }

  this.$get = function(a) {
    return new Greeter(a);
  };
});
```

We can then do this:

```
angular.module('abc', []).config(function(greeter2Provider) {
  greeter2Provider.setSalutation('Halo');
});

function Controller(greeter2) {
  expect(greeter2.greet()).toEqual('Halo 123');
}
```

As a side note, `service`, `factory`, and `value` are all derived from provider.

```
provider.service = function(name, Class) {
  provider.provide(name, function() {
    this.$get = function($injector) {
      return $injector.instantiate(Class);
    };
  });
}

provider.factory = function(name, factory) {
  provider.provide(name, function() {
    this.$get = function($injector) {
      return $injector.invoke(factory);
    };
  });
}

provider.value = function(name, value) {
  provider.factory(name, function() {
    return value;
  });
};
```

edited Jul 19 at 15:52

community wiki
16 revs, 15 users 71%
Lior

32 See also stackoverflow.com/a/13763886/215945 which discusses the differences between service and factory. – Mark Rajcok Mar 27 '13 at 18:10

8 I think there is another mistake in the provider example. greeter.greet() returns 'Halo'. '123' is a parameter. – roland Aug 14 '13 at 12:16

9 Although a service is called by creating an instance of the function. It is actually created only once per injector which makes it like singleton.docs.angularjs.org/guide/dev_guide.services.creating_services – angelokh Dec 15 '13 at 6:17

9 I downvoted this because you never explain what the `provide` object is. As a newbie to angular I'm

pretty sure I can infer what `provide` but since you don't explain it, this answer to clear up some aspects of Angular potentially adds more confusion to the subject. – Spencer Carnage Apr 25 '14 at 22:30

5 This example could be incredible if it used a clear practical example. I get lost trying to figure out what the point of things like `toEqual` and `greeter.Greet` is. Why not use something slightly more real and relatable? – Kyle Pennell Aug 6 '14 at 18:35

Live example

" Hello world " example with `factory` / `service` / `provider` :

```
var myApp = angular.module('myApp', []);

//Service style, probably the simplest one
myApp.service('helloWorldFromService', function() {
    this.sayHello = function() {
        return "Hello, World!"
    };
});

//Factory style, more involved but more sophisticated
myApp.factory('helloWorldFromFactory', function() {
    return {
        sayHello: function() {
            return "Hello, World!"
        }
    };
});

//Provider style, full blown, configurable version
myApp.provider('helloWorld', function() {
    // In the provider function, you cannot inject any
    // service or factory. This can only be done at the
    // "$get" method.

    this.name = 'Default';

    this.$get = function() {
        var name = this.name;
        return {
            sayHello: function() {
                return "Hello, " + name + "!"
            }
        }
    };

    this.setName = function(name) {
        this.name = name;
    };
});

//Hey, we can configure a provider!
myApp.config(function(helloWorldProvider){
    helloWorldProvider.setName('World');
});


function MyCtrl($scope, helloWorld, helloWorldFromFactory, helloWorldFromService) {

    $scope.hellos = [
        helloWorld.sayHello(),
        helloWorldFromFactory.sayHello(),
        helloWorldFromService.sayHello()];
}?
```

edited Nov 29 '14 at 9:05

Peter Mortensen
8,756 10 58 95

answered Jul 30 '13 at 10:20

EpokK
21.5k 4 35 56

11 See it live here: jsfiddle.net/pkozlowski_opensource/PxdSP/14 – smets.kevin Aug 30 '13 at 8:00

1 Doesn't `this` change context in the `$get` function? - you no longer refer to the instantiated provider in that function. – Nate-Wilkins Oct 23 '13 at 19:17

3 @Nate: `this` doesn't change context, actually, because what's being called is `new Provider()` .$get(), where `Provider` is the function being passed to `app.provider` . That is to say that `$get()` is being called as a method on the constructed `Provider` , so `this` will refer to `Provider` as the example suggests. – Brandon Oct 28 '13 at 15:50

@Brandon Ohh ok that's kindof neat then. Confusing at first glance - thanks for the clarification! – Nate-Wilkins Oct 28 '13 at 16:03

1 Is the reason why @Antoine gets the "Unknown provide: helloWorldProvider" error because in your .config code, you use 'helloWorldProvider', but when you define the provider in myApp.provider('helloWorld', function()), you use 'helloWorld'? In other words, in your config code, how does angular know you are referring to the helloWorld provider? Thanks – user3192006 May 3 '14 at 6:48

All Services are **singletons**; they get instantiated once per app. They can be **of any type**, whether it be a primitive, object literal, function, or even an instance of a custom type.

The `value`, `factory`, `service`, `constant`, and `provider` methods are all providers. They teach the Injector how to instantiate the Services.

> The most verbose, but also the most comprehensive one is a Provider recipe. The **remaining four** recipe types — Value, Factory, Service and Constant — **are just syntactic sugar on top of a provider recipe**.

- The **Value Recipe** is the simplest case, where you instantiate the Service yourself and provide the **instantiated value** to the injector.
- The **Factory recipe** gives the Injector a factory function that it calls when it needs to instantiate the service. When called, the **factory function** creates and returns the service instance. The dependencies of the Service are injected as the functions' arguments. So using this recipe adds the following abilities:
  - Ability to use other services (have dependencies)
  - Service initialization
  - Delayed/lazy initialization
- The **Service recipe** is almost the same as the Factory recipe, but here the Injector invokes a **constructor** with the new operator instead of a factory function.
- The **Provider recipe** is usually **overkill**. It adds one more layer of indirection by allowing you to configure the creation of the factory.

  > You should use the Provider recipe only when you want to expose an API for application-wide configuration that must be made before the application starts. This is usually interesting only for reusable services whose behavior might need to vary slightly between applications.

- The **Constant recipe** is just like the Value recipe except it allows you to define services that are available in the **config** phase. Sooner than services created using the Value recipe. Unlike Values, they cannot be decorated using `decorator`.

See the provider documentation.

edited Dec 13 '14 at 17:01
Peter Mortensen
8,756 10 58 95

answered Dec 24 '13 at 13:15
flup
17.7k 5 27 48

Well explained in plain English. I liked your comment on when to use Providers. It would also be very helpful if you could add similar comments for Factory and Service. Thanks! – awavi Jan 4 at 14:03

So service and factory are essentially the same? Using one of the other provides nothing other than alternative syntax? – Matt Feb 21 at 21:36

1 @Matt, yes, service is a concise way when you already have your own function which you want to expose as a service. From docs: myApp.factory('unicornLauncher', ["apiToken", function(apiToken) { return new UnicornLauncher(apiToken); }]); vs: myApp.service('unicornLauncher', ["apiToken", UnicornLauncher]); – janek Feb 22 at 18:39

@joshperry As a newbie, I've googled the difference between service and factory for a while. I do agree this is the best answer ever! I'd understand service as a service class (e.g. encoder/decoder class), which might have some private properties. And factory provides a set of stateless helper methods. – stanleyxu2005 Mar 14 at 15:16

Yaa examples in other answers above fail to very clearly explain the core difference b/w services and providers which is what is injected at the time when these recipes are instantiated. – ashish173 Apr 23 at 7:43

TL;DR

1) When you're using a **Factory** you create an object, add properties to it, then return that same object. When you pass this factory into your controller, those properties on the object will now be available in that controller through your factory.

```
app.controller('myFactoryCtrl', function($scope, myFactory){
  $scope.artist = myFactory.getArtist();
});

app.factory('myFactory', function(){
  var _artist = 'Shakira';
  var service = {};

  service.getArtist = function(){
    return _artist;
  }

  return service;
});
```

2) When you're using **Service**, AngularJS instantiates it behind the scenes with the 'new' keyword. Because of that, you'll add properties to 'this' and the service will return 'this'. When you pass the service into your controller, those properties on 'this' will now be available on that controller through your service.

```
app.controller('myServiceCtrl', function($scope, myService){
  $scope.artist = myService.getArtist();
});

app.service('myService', function(){
  var _artist = 'Nelly';
  this.getArtist = function(){
    return _artist;
  }
});
```

2) Providers are the only service you can pass into your .config() function. Use a provider when you want to provide module-wide configuration for your service object before making it available.

```
app.controller('myProvider', function($scope, myProvider){
  $scope.artist = myProvider.getArtist();
  $scope.data.thingFromConfig = myProvider.thingOnConfig;
});

app.provider('myProvider', function(){
 //Only the next two lines are available in the app.config()
 this._artist = '';
 this.thingFromConfig = '';
  this.$get = function(){
    var that = this;
    return {
      getArtist: function(){
        return that._artist;
      },
      thingOnConfig: that.thingFromConfig
    }
  }
});

app.config(function(myProviderProvider){
  myProviderProvider.thingFromConfig = 'This was set in config';
});
```

Non TL;DR

1) Factory
Factories are the most popular way to create and configure a service. There's really not much more than what the TL;DR said. You just create an object, add properties to it, then return that same object. Then when you pass the factory into your controller, those properties on the object will now be available in that controller through your factory. A more extensive example is below.

```
app.factory('myFactory', function(){
  var service = {};
  return service;
});
```

Now whatever properties we attach to 'service' will be available to us when we pass 'myFactory' into our controller.

Now let's add some 'private' variables to our callback function. These won't be directly accessible from the controller, but we will eventually set up some getter/setter methods on 'service' to be able to alter these 'private' variables when needed.

```
app.factory('myFactory', function($http, $q){
  var service = {};
  var baseUrl = 'https://itunes.apple.com/search?term=';
  var _artist = '';
  var _finalUrl = '';

  var makeUrl = function(){
   _artist = _artist.split(' ').join('+');
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK';
    return _finalUrl
  }

  return service;
});
```

Here you'll notice we're not attaching those variables/function to 'service'. We're simply creating them in order to either use or modify them later.

- baseUrl is the base URL that the iTunes API requires
- _artist is the artist we wish to lookup
- _finalUrl is the final and fully built URL to which we'll make the call to iTunes makeUrl is a

function that will create and return our iTunes friendly URL.

Now that our helper/private variables and function are in place, let's add some properties to the 'service' object. Whatever we put on 'service' we'll be able to directly use in whichever controller we pass 'myFactory' into.

We are going to create setArtist and getArtist methods that simply return or set the artist. We are also going to create a method that will call the iTunes API with our created URL. This method is going to return a promise that will fulfill once the data has come back from the iTunes API. If you haven't had much experience using promises in AngularJS, I highly recommend doing a deep dive on them.

Below **setArtist** accepts an artist and allows you to set the artist. **getArtist** returns the artist callItunes first calls makeUrl() in order to build the URL we'll use with our $http request. Then it sets up a promise object, makes an $http request with our final url, then because $http returns a promise, we are able to call .success or .error after our request. We then resolve our promise with the iTunes data, or we reject it with a message saying 'There was an error'.

```
app.factory('myFactory', function($http, $q){
  var service = {};
  var baseUrl = 'https://itunes.apple.com/search?term=';
  var _artist = '';
  var _finalUrl = '';

  var makeUrl = function(){
    _artist = _artist.split(' ').join('+');
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK'
    return _finalUrl;
  }

  service.setArtist = function(artist){
    _artist = artist;
  }

  service.getArtist = function(){
    return _artist;
  }

  service.callItunes = function(){
    makeUrl();
    var deferred = $q.defer();
    $http({
      method: 'JSONP',
      url: _finalUrl
    }).success(function(data){
      deferred.resolve(data);
    }).error(function(){
      deferred.reject('There was an error')
    })
    return deferred.promise;
  }

  return service;
});
```

Now our factory is complete. We are now able to inject 'myFactory' into any controller and we'll then be able to call our methods that we attached to our service object (setArtist, getArtist, and callItunes).

```
app.controller('myFactoryCtrl', function($scope, myFactory){
  $scope.data = {};
  $scope.updateArtist = function(){
    myFactory.setArtist($scope.data.artist);
  };

  $scope.submitArtist = function(){
    myFactory.callItunes()
      .then(function(data){
        $scope.data.artistData = data;
      }, function(data){
        alert(data);
      })
  }
});
```

In the controller above we're injecting in the 'myFactory' service. We then set properties on our $scope object that are coming from data from 'myFactory'. The only tricky code above is if you've never dealt with promises before. Because callItunes is returning a promise, we are able to use the .then() method and only set $scope.data.artistData once our promise is fulfilled with the iTunes data. You'll notice our controller is very 'thin'. All of our logic and persistent data is located in our service, not in our controller.

2) Service

Perhaps the biggest thing to know when dealing with creating a Service is that that it's instantiated with the 'new' keyword. For you JavaScript gurus this should give you a big hint into the nature of the code. For those of you with a limited background in JavaScript or for those who aren't too familiar with what the 'new' keyword actually does, let's review some JavaScript fundamentals that will eventually help us in understanding the nature of a Service.

To really see the changes that occur when you invoke a function with the 'new' keyword, let's

create a function and invoke it with the 'new' keyword, then let's show what the interpreter does when it sees the 'new' keyword. The end results will both be the same.

First let's create our Constructor.

```
var Person = function(name, age){
  this.name = name;
  this.age = age;
}
```

This is a typical JavaScript constructor function. Now whenever we invoke the Person function using the 'new' keyword, 'this' will be bound to the newly created object.

Now let's add a method onto our Person's prototype so it will be available on every instance of our Person 'class'.

```
Person.prototype.sayName = function(){
  alert('My name is ' + this.name);
}
```

Now, because we put the sayName function on the prototype, every instance of Person will be able to call the sayName function in order alert that instance's name.

Now that we have our Person constructor function and our sayName function on its prototype, let's actually create an instance of Person then call the sayName function.

```
var tyler = new Person('Tyler', 23);
tyler.sayName(); //alerts 'My name is Tyler'
```

So all together the code for creating a Person constructor, adding a function to it's prototype, creating a Person instance, and then calling the function on its prototype looks like this.

```
var Person = function(name, age){
  this.name = name;
  this.age = age;
}
Person.prototype.sayName = function(){
  alert('My name is ' + this.name);
}
var tyler = new Person('Tyler', 23);
tyler.sayName(); //alerts 'My name is Tyler'
```

Now let's look at what actually is happening when you use the 'new' keyword in JavaScript. First thing you should notice is that after using 'new' in our example, we're able to call a method (sayName) on 'tyler' just as if it were an object - that's because it is. So first, we know that our Person constructor is returning an object, whether we can see that in the code or not. Second, we know that because our sayName function is located on the prototype and not directly on the Person instance, the object that the Person function is returning must be delegating to its prototype on failed lookups. In more simple terms, when we call tyler.sayName() the interpreter says "OK, I'm going to look on the 'tyler' object we just created, locate the sayName function, then call it. Wait a minute, I don't see it here - all I see is name and age, let me check the prototype. Yup, looks like it's on the prototype, let me call it.".

Below is code for how you can think about what the 'new' keyword is actually doing in JavaScript. It's basically a code example of the above paragraph. I've put the 'interpreter view' or the way the interpreter sees the code inside of notes.

```
var Person = function(name, age){
  //The line below this creates an obj object that will delegate to the person's prototype
on failed lookups.
  //var obj = Object.create(Person.prototype);

  //The line directly below this sets 'this' to the newly created object
  //this = obj;

  this.name = name;
  this.age = age;

  //return this;
}
```

Now having this knowledge of what the 'new' keyword really does in JavaScript, creating a Service in AngularJS should be easier to understand.

The biggest thing to understand when creating a Service is knowing that Services are instantiated with the 'new' keyword. Combining that knowledge with our examples above, you should now recognize that you'll be attaching your properties and methods directly to 'this' which will then be returned from the Service itself. Let's take a look at this in action.

Unlike what we originally did with the Factory example, we don't need to create an object then return that object because, like mentioned many times before, we used the 'new' keyword so the interpreter will create that object, have it delegate to it's prototype, then return it for us without us having to do the work.

First things first, let's create our 'private' and helper function. This should look very familiar since we did the exact same thing with our factory. I won't explain what each line does here because I did that in the factory example, if you're confused, re-read the factory example.

```
app.service('myService', function($http, $q){
  var baseUrl = 'https://itunes.apple.com/search?term=';
  var _artist = '';
  var _finalUrl = '';

  var makeUrl = function(){
    _artist = _artist.split(' ').join('+');
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK'
    return _finalUrl;
  }
});
```

Now, we'll attach all of our methods that will be available in our controller to 'this'.

```
app.service('myService', function($http, $q){
  var baseUrl = 'https://itunes.apple.com/search?term=';
  var _artist = '';
  var _finalUrl = '';

  var makeUrl = function(){
    _artist = _artist.split(' ').join('+');
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK'
    return _finalUrl;
  }

  this.setArtist = function(artist){
    _artist = artist;
  }

  this.getArtist = function(){
    return _artist;
  }

  this.callItunes = function(){
    makeUrl();
    var deferred = $q.defer();
    $http({
      method: 'JSONP',
      url: _finalUrl
    }).success(function(data){
      deferred.resolve(data);
    }).error(function(){
      deferred.reject('There was an error')
    })
    return deferred.promise;
  }

});
```

Now just like in our factory, setArtist, getArtist, and callItunes will be available in whichever controller we pass myService into. Here's the myService controller (which is almost exactly the same as our factory controller).

```
app.controller('myServiceCtrl', function($scope, myService){
  $scope.data = {};
  $scope.updateArtist = function(){
    myService.setArtist($scope.data.artist);
  };

  $scope.submitArtist = function(){
    myService.callItunes()
      .then(function(data){
        $scope.data.artistData = data;
      }, function(data){
        alert(data);
      })
  }
});
```

Like I mentioned before, once you really understand what 'new' does, Services are almost identical to factories in AngularJS.

3) Provider

The biggest thing to remember about Providers is that they're the only service that you can pass into the app.config portion of your application. This is of huge importance if you're needing to alter some portion of your service object before it's available everywhere else in your application. Although very similar to Services/Factories, there are a few differences which we'll discuss.

First we set up our Provider in a similar way we did with our Service and Factory. The variables below are our 'private' and helper function.

```
app.provider('myProvider', function(){
   var baseUrl = 'https://itunes.apple.com/search?term=';
  var _artist = '';
  var _finalUrl = '';

  //Going to set this property on the config function below.
  this.thingFromConfig = '';

  var makeUrl = function(){
    _artist = _artist.split(' ').join('+');
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK'
    return _finalUrl;
```

```
    }
}
```

*Again if any portion of the above code is confusing, check out the Factory section where I explain what it all does it greater details.

You can think of Providers as having three sections. The first section is the 'private' variables/functions that will be modified/set later (shown above). The second section is the variables/functions that will be available in your app.config function and are therefore available to alter before they're available anywhere else (also shown above). It's important to note that those variables need to be attached to the 'this' keyword. In our example, only 'thingFromConfig' will be available to alter in the app.config. The third section (shown below) is all the variables/functions that will be available in your controller when you pass in the 'myProvider' service into that specific controller.

When creating a service with Provider, the only properties/methods that will be available in your controller are those properties/methods which are returned from the $get() function. The code below puts $get on 'this' (which we know will eventually be returned from that function). Now, that $get function returns all the methods/properties we want to be available in the controller. Here's a code example.

```
this.$get = function($http, $q){
    return {
      callItunes: function(){
        makeUrl();
        var deferred = $q.defer();
        $http({
          method: 'JSONP',
          url: _finalUrl
        }).success(function(data){
          deferred.resolve(data);
        }).error(function(){
          deferred.reject('There was an error')
        })
        return deferred.promise;
      },
      setArtist: function(artist){
        _artist = artist;
      },
      getArtist: function(){
        return _artist;
      },
      thingOnConfig: this.thingFromConfig
    }
  }
```

Now the full Provider code looks like this

```
app.provider('myProvider', function(){
  var baseUrl = 'https://itunes.apple.com/search?term=';
  var _artist = '';
  var _finalUrl = '';

  //Going to set this property on the config function below
  this.thingFromConfig = '';

  var makeUrl = function(){
    _artist = _artist.split(' ').join('+');
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK'
    return _finalUrl;
  }

  this.$get = function($http, $q){
    return {
      callItunes: function(){
        makeUrl();
        var deferred = $q.defer();
        $http({
          method: 'JSONP',
          url: _finalUrl
        }).success(function(data){
          deferred.resolve(data);
        }).error(function(){
          deferred.reject('There was an error')
        })
        return deferred.promise;
      },
      setArtist: function(artist){
        _artist = artist;
      },
      getArtist: function(){
        return _artist;
      },
      thingOnConfig: this.thingFromConfig
    }
  }
});
```

Now just like in our factory and Service, setArtist, getArtist, and callItunes will be available in whichever controller we pass myProvider into. Here's the myProvider controller (which is almost exactly the same as our factory/Service controller).

```
app.controller('myProviderCtrl', function($scope, myProvider){
```

```
  $scope.data = {};
  $scope.updateArtist = function(){
    myProvider.setArtist($scope.data.artist);
  };

  $scope.submitArtist = function(){
    myProvider.callItunes()
      .then(function(data){
        $scope.data.artistData = data;
      }, function(data){
        alert(data);
      })
  }

  $scope.data.thingFromConfig = myProvider.thingOnConfig;
});
```

As mentioned before, the whole point of creating a service with Provider is to be able to alter some variables through the app.config function before the final object is passed to the rest of the application. Let's see an example of that.

```
app.config(function(myProviderProvider){
  //Providers are the only service you can pass into app.config
  myProviderProvider.thingFromConfig = 'This sentence was set in app.config. Providers are
the only service that can be passed into config. Check out the code to see how it works';
});
```

Now you can see how 'thingFromConfig' is as empty string in our provider, but when that shows up in the DOM, it will be 'This sentence was set…'.

edited Dec 13 '14 at 17:09
Peter Mortensen
8,756 10 58 95

answered May 15 '14 at 15:53
Tyler McGinnis
6,478 3 14 28

2 The only part that is missing in this excellent write-up is the relative advantages of using the service over an factory; which is clearly explained in the the accepted answer by Lior – iNfinity Oct 8 '14 at 4:46

Awesome explanation!! @tyler Could you just add one thing? How to decide between using the service or the factory? I see how they are different but when using them I feel like it changes nothing so why go with one and not the other? Thanks!!! – commandantp Jan 28 at 13:02

FWIW(maybe not much), here is a blogger that takes issue with Angular, and doesn't like providerProvider codeofrob.com/entries/you-have-ruined-javascript.html – barlop Aug 6 at 21:08

For me, the revelation came when I realized that they all work the same way: by running something **once**, storing the value they get, and then cough up **that same stored value** when referenced through dependency injection.

Say we have:

```
app.factory('a', fn);
app.service('b', fn);
app.provider('c', fn);
```

The difference between the three is that:

1. `a` 's stored value comes from running `fn` .
2. `b` 's stored value comes from `new` ing `fn` .
3. `c` 's stored value comes from first getting an instance by `new` ing `fn` , and then running a `$get` method of the instance.

Which means there's something like a cache object inside AngularJS, whose value of each injection is only assigned once, when they've been injected the first time, and where:

```
cache.a = fn()
cache.b = new fn()
cache.c = (new fn()).$get()
```

This is why we use `this` in services, and define a `this.$get` in providers.

edited Dec 13 '14 at 18:14
Peter Mortensen
8,756 10 58 95

answered Nov 14 '14 at 6:25
Luxiyalu
2,356 1 8 19

Understanding AngularJS Factory, Service and Provider

All of these are used to share reusable singleton objects. It helps to share reusable code across your app/various components/modules.

> From Docs **Service/Factory**:
>
> - **Lazily instantiated** – Angular only instantiates a service/factory when an application

component depends on it.

- **Singletons** – Each component dependent on a service gets a reference to the single instance generated by the service factory.

Factory

A factory is function where you can manipulate/add logic before creating an object, then the newly created object gets returned.

```
app.factory('MyFactory', function() {
    var serviceObj = {};
    //creating an object with methods/functions or variables
    serviceObj.myFunction = function() {
        //TO DO:
    };
    //return that object
    return serviceObj;
});
```

Usage

It can be just a collection of functions like a class. Hence, it can be instantiated in different controllers when you are injecting it inside your controller/factory/directive functions. It is instantiated only once per app.

Service

Simply while looking at the services think about the array prototype. A service is a function which instantiates a new object using the 'new' keyword. You can add properties and functions to a service object by using the `this` keyword. Unlike a factory, it doesn't return anything (it returns an object which contains methods/properties).

```
app.service('MyService', function() {
    //directly binding events to this context
    this.myServiceFunction = function() {
        //TO DO:
    };
});
```

Usage

Use it when you need to share a single object throughout the application. For example, authenticated user details, share-able methods/data, Utility functions etc.

Provider

A provider is used to create a configurable service object. You can configure the service setting from config function. It returns a value by using the `$get()` function. The `$get` function gets executed on the run phase in angular.

```
app.provider('configurableService', function() {
    var name = '';
    //this method can be be available at configuration time inside app.config.
    this.setName = function(newName) {
        name = newName;
    };
    this.$get = function() {
        var myFunction = function() {
            return 'Some Name'
        }
        return {
            myFunction: myFunction //exposed object to where it gets injected.
        };
    };
});
```

Usage

When you need to provide module-wise configuration for your service object before making it available, eg. suppose you want to set your API URL on basis of your Environment like `dev`, `stage` or `prod`

> **NOTE**
>
> Only provider will be available in config phase of angular, while service & factory are not.

Hope this has cleared up your understanding about **Factory, Service and Provider**.

edited Sep 3 at 17:32
lmyers
1,288 7 15

answered Feb 1 at 12:58
Pankaj Parkar
28.8k 5 17 51

1 Great answer, possibly the clearest here. – Mrk Fldig Jun 16 at 8:44

@MrkFldig Glad to know that you liked it & understood well..Thanks :) – Pankaj Parkar Jun 16 at 10:42

1 Very easy to understand – Tech Solvr Jul 26 at 13:23

1 @TechSolvr Good to hear that.. :) – Pankaj Parkar Jul 26 at 13:25

What would I do if I wanted to have a service with a particular interface, but have two different implementations, and inject each into a controller but tied to different states using ui-router? e.g. make remote calls in one state, but write to local storage instead in another. The provider docs say to use `only when you want to expose an API for application-wide configuration that must be made before the application starts. This is usually interesting only for reusable services whose behavior might need to vary slightly between applications` , so doesn't sound possible, right? – qix Sep 9 at 0:37

Service vs provider vs factory:

I am trying to keep it simple. It's all about a basic JavaScript concept.

First of all, let's talk about **services** in AngularJS!

What is Service: In AngularJS, **Service** is nothing but a singleton JavaScript object which can store some useful methods or properties. This singleton object is created per ngApp(Angular app) basis and it is shared among all the controllers within current app. When Angularjs instantiate a service object, it register this service object with a unique service name. So each time when we need to get the service instance, Angular search the registry for this service name, and it returns the reference to service object. Now we can invoke method, access properties on service object. You may have question whether you can also put properties, methods on scope object of controllers! So why you need service object? Answers is: services are shared among multiple controller scope. If you put some properties/methods in a controller's scope object , it will be available to current controller scope only. But when you define methods,properties on service object, it will be available globally for every controller's scope for current AngularJS app.

So if there are three controllers scope, let it be controllerA, controllerB and controllerC, all will share the same service instance.

```
<div ng-controller='controllerA'>
    <!-- controllerA scope -->
</div>
<div ng-controller='controllerB'>
    <!-- controllerB scope -->
</div>
<div ng-controller='controllerC'>
    <!-- controllerC scope -->
</div>
```

How to create a service?

AngularJS provide different methods to registering a service. Here we are going to focus on three methods factory(..),service(..),provide(..);

Use this link for code reference

Factory function:

We can define a factory function as below.

```
factory('serviceName',function fnFactory(){ return serviceInstance;})
```

AngularJS provides the factory() method which takes a JavaScript function as a parameter. Angular calls this passed function **fnFactory()** for creating service instance like below.

```
var serviceInstace = fnFactory();
```

The passed function can define a object and return that object, AngularJS simply stores this object reference to a variable which is passed as first argument. Whatever we put on this object , will be available to service instance. Instead of returning object , we can also return function, values etc, Whatever we will return , will be available to service instance.

Example:

```
var app= angular.module('myApp', []);
//creating service using factory method
app.factory('factoryPattern',function(){
  var data={
    'firstName':'Tom',
    'lastName':' Cruise',
    greet: function(){
      console.log('hello!' + this.firstName + this.lastName);
    }
  };

  //Now all the properties and methods of data object will be available in our service
object
  return data;
});
```

Service Function:

```
service('serviceName',function fnServiceConstructor(){})
```

It's the another way, we can register a service. The only difference is the way AngularJS tries to instantiate the service object. This time angular uses 'new' keyword and call the constructor function something like below.

```
var serviceInstance = new fnServiceConstructor();
```

In the constructor function we can use 'this' keyword for adding properties/methods to the service object. example:

```
//Creating a service using the service method
var app= angular.module('myApp', []);
app.service('servicePattern',function(){
  this.firstName ='James';
  this.lastName =' Bond';
  this.greet = function(){
    console.log('My Name is '+ this.firstName + this.lastName);
  };
});
```

Provide function:

Provide() function is the another way for creating services. Let we are interested to create a service which just display some greeting message to the user. But we also want to provide a way so that user can set their own greeting message. In technical terms we want to create configurable services. How can we do this ? There must be a way, so that app could pass their custom greeting messages and Angularjs would make it available to factory/constructor function which create our services instance. In such a case provide() function do the job. using provide() function we can create configurable services.

We can create configurable services using provide syntax as given below.

```
/*step1:define a service */
app.provide('service',function serviceProviderConstructor(){});

/*step2:configure the service */
app.config(function configureService(serviceProvider){});
```

How does provider syntax internally work?

1.Provider object is created using constructor function we defined in our provide function.

```
var serviceProvider = new serviceProviderConstructor();
```

2.The function we passed in app.config(), get executed. This is called config phase, and here we have a chance to customize our service.

```
configureService(serviceProvider);
```

3.Finally service instance is created by calling $get method of serviceProvider.

```
serviceInstance = serviceProvider.$get()
```

Sample code for creating service using provide syntax:

```
var app= angular.module('myApp', []);
app.provider('providerPattern',function providerConstructor(){
  //this function works as constructor function for provider
  this.firstName = 'Arnold ';
  this.lastName = ' Schwarzenegger' ;
  this.greetMessage = ' Welcome, This is default Greeting Message' ;
  //adding some method which we can call in app.config() function
  this.setGreetMsg = function(msg){
    if(msg){
      this.greetMessage =  msg ;
    }
  };

  //We can also add a method which can change firstName and lastName
  this.$get = function(){
    var firstName = this.firstName;
    var lastName = this.lastName ;
    var greetMessage = this.greetMessage;
    var data={
       greet: function(){
         console.log('hello, ' + firstName + lastName+'! '+ greetMessage);
       }
    };
    return data ;
  };
});

app.config(
  function(providerPatternProvider){
    providerPatternProvider.setGreetMsg(' How do you do ?');
  }
);
```

Working Demo

Summary:

Factory use a factory function which return a service instance. **serviceInstance = fnFactory();**

Service use a constructor function and Angular invoke this constructor function using 'new' keyword for creating the service instance. **serviceInstance = new fnServiceConstructor();**

Provide defines a providerConstructor function, this providerConstructor function defines a factory function **$get** . Angular calls $get() to create the service object. Provide syntax has an added advantage of configuring the service object before it get instantiated. **serviceInstance = $get();**

edited Dec 13 '14 at 17:16
Peter Mortensen
8,756 10 58 95

answered Jul 22 '14 at 11:39
Anant
1,294 1 8 14

1 Great explanation! – Bram Oct 17 '14 at 9:49

1 Brilliant on detailing. – ruionwriting May 7 at 10:12

Factory

You give AngularJS a function, AngularJS will cache and inject the return value when the factory is requested.

Example:

```
app.factory('factory', function() {
    var name = '';
    // Return value **is** the object that will be injected
    return {
        name: name;
    }
})
```

Usage:

```
app.controller('ctrl', function($scope, factory) {
     $scope.name = factory.name;
});
```

Service

You give AngularJS a function, AngularJS will call ***new*** to instantiate it. It is the instance that AngularJS creates that will be cached and injected when the service is requested. Since ***new*** was used to instantiate the service, the keyword ***this*** is valid and refers to the instance.

Example:

```
app.service('service', function() {
     var name = '';
     this.setName = function(newName) {
         name = newName;
     }
     this.getName = function() {
         return name;
     }
});
```

Usage:

```
app.controller('ctrl', function($scope, service) {
   $scope.name = service.getName();
});
```

Provider

You give AngularJS a function, and AngularJS will call its `$get` function. It is the return value from the `$get` function that will be cached and injected when the service is requested.

Providers allow you to configure the provider ***before*** AngularJS calls the `$get` method to get the injectible.

Example:

```
app.provider('provider', function() {
     var name = '';
     this.setName = function(newName) {
          name = newName;
     }
     this.$get = function() {
```

```
        return {
           name: name
        }
    }
})
```

Usage (as an injectable in a controller)

```
app.controller('ctrl', function($scope, provider) {
    $scope.name = provider.name;
});
```

Usage (configuring the provider before `$get` is called to create the injectable)

```
app.config(function(providerProvider) {
    providerProvider.setName('John');
});
```

edited Dec 13 '14 at 17:19
Peter Mortensen
8,756 10 58 95

answered Aug 2 '14 at 5:37
pixelbits
11.9k 2 14 31

I noticed something interesting when playing around with providers.

Visibility of injectables is different for providers than it is for services and factories. If you declare an AngularJS "constant" (for example, `myApp.constant('a', 'Robert');`), you can inject it into services, factories, and providers.

But if you declare an AngularJS "value" (for example., `myApp.value('b', {name: 'Jones'});`), you can inject it into services and factories, but NOT into the provider-creating function. You can, however, inject it into the `$get` function that you define for your provider. This is mentioned in the AngularJS documentation, but it's easy to miss. You can find it on the %provide page in the sections on the value and constant methods.

http://jsfiddle.net/R2Frv/1/

```
<div ng-app="MyAppName">
    <div ng-controller="MyCtrl">
        <p>from Service: {{servGreet}}</p>
        <p>from Provider: {{provGreet}}</p>
    </div>
</div>
<script>
    var myApp = angular.module('MyAppName', []);

    myApp.constant('a', 'Robert');
    myApp.value('b', {name: 'Jones'});

    myApp.service('greetService', function(a,b) {
        this.greeter = 'Hi there, ' + a + ' ' + b.name;
    });

    myApp.provider('greetProvider', function(a) {
        this.firstName = a;
        this.$get = function(b) {
            this.lastName = b.name;
            this.fullName = this.firstName + ' ' + this.lastName;
            return this;
        };
    });

    function MyCtrl($scope, greetService, greetProvider) {
        $scope.servGreet = greetService.greeter;
        $scope.provGreet = greetProvider.fullName;
    }
</script>
```

edited Apr 21 at 18:55
mohammadreza berneti
227 3 11

answered May 19 '13 at 19:53
justlooking
291 2 3

1 why this strange behavior? – suzanshakya Oct 20 '13 at 19:14

For me the best and the simplest way of understanding the difference is:

```
var service, factory;
service = factory = function(injection) {}
```

How AngularJS instantiates particular components (simplified):

```
// service
var angularService = new service(injection);

// factory
var angularFactory = factory(injection);
```

So, for the service, what becomes the AngularJS component is the object instance of the class which is represented by service declaration function. For the factory, it is the result returned from the factory declaration function. The factory may behave the same as the service:

```
var factoryAsService = function(injection) {
  return new function(injection) {
    // Service content
  }
}
```

The simplest way of thinking is the following one:

- Service is an singleton object instance. Use services if you want to provide a singleton object for your code.
- Factory is a class. Use factories if you want to provide custom classes for your code (can't be done with services because they are already instantiated).

The factory 'class' example is provided in the comments around, as well as provider difference.

edited Dec 13 '14 at 17:03
Peter Mortensen
8,756 10 58 95

answered Apr 30 '14 at 11:20
l0co
958 7 11

Please consider using this as a cheatsheet while learning AngularJS: http://demisx.github.io/angularjs/2014/09/14/angular-what-goes-where.html. This should provide a direction on what to place where in AngularJS.

edited Dec 13 '14 at 17:20
Peter Mortensen
8,756 10 58 95

answered Sep 16 '14 at 17:01
demisx
789 7 13

great work, thanks – Allisone Sep 16 '14 at 21:19

My clarification on this matter:

Basically all of the mentioned types (service, factory, provider, etc.) are just creating and configuring global variables (that are of course global to the entire application), just as old fashioned global variables were.

While global variables are not recommended, the real usage of these global variables is to provide dependency injection, by passing the variable to the relevant controller.

There are many levels of complications in creating the values for the "global variables":

1. **Constant**
   This defines an actual constant that should not be modified during the entire application, just like constants in other languages are (something that JavaScript lacks).
2. **Value**
   This is a modifiable value or object, and it serves as some global variable, that can even be injected when creating other services or factories (see further on these). However, it must be a "*literal value*", which means that one has to write out the actual value, and cannot use any computation or programming logic (in other words *39* or *myText* or *{prop: "value"}* are OK, but *2 +2* is not).
3. **Factory**
   A more general value, that is possible to be computed right away. It works by passing a function to AngularJS with the logic needed to compute the value and AngularJS executes it, and it saves the return value in the named variable.
   Note that it is possible to return a object (in which case it will function similar to a *service*) or a function (that will be saved in the variable as a callback function).
4. **Service**
   A service is a more stripped-down version of *factory* which is valid only when the value is an object, and it allows for writing any logic directly in the function (as if it would be a constructor), as well as declaring and accessing the object properties using the *this* keyword.
5. **Provider**
   Unlike a service which is a simplified version of *factory*, a provider is a more complex, but more flexible way of initializing the "global" variables, with the biggest flexibility being the option to set values from the app.config.
   It works like using a combination of *service* and *provider*, by passing to provider a function that has properties declared using the *this* keyword, which can be used from the `app.config` .
   Then it needs to have a separate *$.get* function which is executed by AngularJS after setting the above properties via the `app.config` file , and this *$.get* function behaves just as the *factory* above, in that its return value is used to initialize the "global" variables.

edited Dec 13 '14 at 18:25


Peter Mortensen
8,756 10 58 95

answered Nov 21 '14 at 0:59


yoel halb
3,800 2 28 27

My understanding is very simple below.

Factory: You simply create an object inside of the factory and return it.

Service:

You just have a standard function that uses this keyword to define a function.

Provider:

There is a `$get` object that you define and it can be used to get the object that returns data.

edited Dec 13 '14 at 17:11


Peter Mortensen
8,756 10 58 95

answered Jun 5 '14 at 7:18


sajan kumar
140 1 6

Didn't you mixed up the Factory and Service ? Services creates where factory returns. – Flavien Volken Nov 9 '14 at 6:06

When you declaring service name as an injectable argument you will be provided with an instance of the function. In other words new FunctionYouPassedToService(). This object instance becomes the service object that AngularJS registers and injects later to other services / controllers if required. //factory When you declaring factoryname as an injectable argument you will be provided with the value that is returned by invoking the function reference passed to module.factory. – sajan kumar Nov 23 '14 at 4:31

Okay, so... in angular the factory is a singleton where the "service" is a actually a factory (in common design patterns terms) – Flavien Volken Nov 23 '14 at 8:24

An additional clarification is that factories can create functions/primitives, while services cannot. Check out this jsFiddle based on Epokk's: http://jsfiddle.net/skeller88/PxdSP/1351/.

The factory returns a function that can be invoked:

```
myApp.factory('helloWorldFromFactory', function() {
  return function() {
    return "Hello, World!";
  };
});
```

The factory can also return an object with a method that can be invoked:

```
myApp.factory('helloWorldFromFactory', function() {
  return {
    sayHello: function() {
      return "Hello, World!";
    }
  };
});
```

The service returns an object with a method that can be invoked:

```
myApp.service('helloWorldFromService', function() {
  this.sayHello = function() {
     return "Hello, World!";
  };
});
```

For more details, see a post I wrote on the difference: http://www.shanemkeller.com/tldr-services-vs-factories-in-angular/

edited Jan 31 at 10:22

answered Apr 30 '14 at 17:20


skeller88
106 5

Factory: The factory you actually create an object inside of the factory and return it.
service: The service you just have a standard function that uses the this keyword to define function.
provider: The provider there's a $get you define and it can be used to get the object that returns the data.

answered Mar 4 at 11:12


MR Kesavan
358 4 11

Using as reference this page and the documentation (which seems to have greatly improved since the last time I looked), I put together the following real(-ish) world demo which uses 4 of the 5 flavours of provider; Value, Constant, Factory and full blown Provider.

HTML:

```
<div ng-controller="mainCtrl as main">
    <h1>{{main.title}}*</h1>
    <h2>{{main.strapline}}</h2>
    <p>Earn {{main.earn}} per click</p>
    <p>You've earned {{main.earned}} by clicking!</p>
    <button ng-click="main.handleClick()">Click me to earn</button>
    <small>* Not actual money</small>
</div>
```

app

```
var app = angular.module('angularProviders', []);

// A CONSTANT is not going to change
app.constant('range', 100);

// A VALUE could change, but probably / typically doesn't
app.value('title', 'Earn money by clicking');
app.value('strapline', 'Adventures in ng Providers');

// A simple FACTORY allows us to compute a value @ runtime.
// Furthermore, it can have other dependencies injected into it such
// as our range constant.
app.factory('random', function randomFactory(range) {
    // Get a random number within the range defined in our CONSTANT
    return Math.random() * range;
});

// A PROVIDER, must return a custom type which implements the functionality
// provided by our service (see what I did there?).
// Here we define the constructor for the custom type the PROVIDER below will
// instantiate and return.
var Money = function(locale) {

    // Depending on locale string set during config phase, we'll
    // use different symbols and positioning for any values we
    // need to display as currency
    this.settings = {
        uk: {
            front: true,
            currency: '£',
            thousand: ',',
            decimal: '.'
        },
        eu: {
            front: false,
            currency: '€',
            thousand: '.',
            decimal: ','
        }
    };

    this.locale = locale;
};

// Return a monetary value with currency symbol and placement, and decimal
// and thousand delimiters according to the locale set in the config phase.
Money.prototype.convertValue = function(value) {

    var settings = this.settings[this.locale],
        decimalIndex, converted;

    converted = this.addThousandSeparator(value.toFixed(2), settings.thousand);

    decimalIndex = converted.length - 3;

    converted = converted.substr(0, decimalIndex) +
        settings.decimal +
        converted.substr(decimalIndex + 1);

    converted = settings.front ?
            settings.currency + converted :
            converted + settings.currency;

    return converted;
};

// Add supplied thousand separator to supplied value
Money.prototype.addThousandSeparator = function(value, symbol) {
   return value.toString().replace(/\B(?=(\d{3})+(?!\d))/g, symbol);
};

// PROVIDER is the core recipe type - VALUE, CONSTANT, SERVICE & FACTORY
// are all effectively syntactic sugar built on top of the PROVIDER construct
// One of the advantages of the PROVIDER is that we can configure it before the
// application starts (see config below).
app.provider('money', function MoneyProvider() {

    var locale;

    // Function called by the config to set up the provider
```

```
    this.setLocale = function(value) {
        locale = value;
    };

    // All providers need to implement a $get method which returns
    // an instance of the custom class which constitutes the service
    this.$get = function moneyFactory() {
        return new Money(locale);
    };
});

// We can configure a PROVIDER on application initialisation.
app.config(['moneyProvider', function(moneyProvider) {
    moneyProvider.setLocale('uk');
    //moneyProvider.setLocale('eu');
}]);

// The ubiquitous controller
app.controller('mainCtrl', function($scope, title, strapline, random, money) {

    // Plain old VALUE(s)
    this.title = title;
    this.strapline = strapline;

    this.count = 0;

    // Compute values using our money provider
    this.earn = money.convertValue(random); // random is computed @ runtime
    this.earned = money.convertValue(0);

    this.handleClick = function() {
        this.count ++;
        this.earned = money.convertValue(random * this.count);
    };
});
```

Working demo.

answered Jan 30 at 20:04

net.uk.sweet
10.2k 2 10 30

---

https://docs.angularjs.org/guide/providers This should probably be the canonical source for the answer to this question. It does a great job of explaining it and it will be kept up to date

answered Feb 16 at 7:52

meffect
485 6 9

Here is some broilerplate code I've come up with as a code-template for object factory in AngularjS. I've used a Car/CarFactory as an example to illustrate. Makes for simple implementation code in the controller.

```
<script>
   angular.module('app', [])
       .factory('CarFactory', function() {

           /**
            * BroilerPlate Object Instance Factory Definition / Example
            */
           this.Car = function() {

               // initialize instance properties
               angular.extend(this, {
                   color           : null,
                   numberOfDoors   : null,
                   hasFancyRadio   : null,
                   hasLeatherSeats : null
               });

               // generic setter (with optional default value)
               this.set = function(key, value, defaultValue, allowUndefined) {

                   // by default,
                   if (typeof allowUndefined === 'undefined') {
                       // we don't allow setter to accept "undefined" as a value
                       allowUndefined = false;
                   }
                   // if we do not allow undefined values, and..
                   if (!allowUndefined) {
                       // if an undefined value was passed in
                       if (value === undefined) {
                           // and a default value was specified
                           if (defaultValue !== undefined) {
                               // use the specified default value
                               value = defaultValue;
                           } else {
                               // otherwise use the class.prototype.defaults value
                               value = this.defaults[key];
                           } // end if/else
```

```
                        } // end if
                    } // end if

                    // update
                    this[key] = value;

                    // return reference to this object (fluent)
                    return this;

                }; // end this.set()

            }; // end this.Car class definition

            // instance properties default values
            this.Car.prototype.defaults = {
                color: 'yellow',
                numberOfDoors: 2,
                hasLeatherSeats: null,
                hasFancyRadio: false
            };

            // instance factory method / constructor
            this.Car.prototype.instance = function(params) {
                return new
                    this.constructor()
                            .set('color',           params.color)
                            .set('numberOfDoors',   params.numberOfDoors)
                            .set('hasFancyRadio',   params.hasFancyRadio)
                            .set('hasLeatherSeats', params.hasLeatherSeats)
                ;
            };

            return new this.Car();

        }) // end Factory Definition
        .controller('testCtrl', function($scope, CarFactory) {

            window.testCtrl = $scope;

            // first car, is red, uses class default for:
            // numberOfDoors, and hasLeatherSeats
            $scope.car1     = CarFactory
                                .instance({
                                    color: 'red'
                                })
                            ;

            // second car, is blue, has 3 doors,
            // uses class default for hasLeatherSeats
            $scope.car2     = CarFactory
                                .instance({
                                    color: 'blue',
                                    numberOfDoors: 3
                                })
                            ;
            // third car, has 4 doors, uses class default for
            // color and hasLeatherSeats
            $scope.car3     = CarFactory
                                .instance({
                                    numberOfDoors: 4
                                })
                            ;
            // sets an undefined variable for 'hasFancyRadio',
            // explicitly defines "true" as default when value is undefined
            $scope.hasFancyRadio = undefined;
            $scope.car3.set('hasFancyRadio', $scope.hasFancyRadio, true);

            // fourth car, purple, 4 doors,
            // uses class default for hasLeatherSeats
            $scope.car4     = CarFactory
                                .instance({
                                    color: 'purple',
                                    numberOfDoors: 4
                                });
            // and then explicitly sets hasLeatherSeats to undefined
            $scope.hasLeatherSeats = undefined;
            $scope.car4.set('hasLeatherSeats', $scope.hasLeatherSeats, undefined,
true);

            // in console, type window.testCtrl to see the resulting objects

        });
    </script>
```

Here is a simpler example. I'm using a few third party libraries that expect a "Position" object exposing latitude and longitude, but via different object properties. I didn't want to hack the vendor code, so I adjusted the "Position" objects I was passing around.

```
    angular.module('app')
.factory('PositionFactory', function() {

    /**
     * BroilerPlate Object Instance Factory Definition / Example
     */
    this.Position = function() {

        // initialize instance properties
        // (multiple properties to satisfy multiple external interface contracts)
```

```
        angular.extend(this, {
            lat         : null,
            lon         : null,
            latitude    : null,
            longitude   : null,
            coords: {
                latitude: null,
                longitude: null
            }
        });

        this.setLatitude = function(latitude) {
            this.latitude           = latitude;
            this.lat                = latitude;
            this.coords.latitude    = latitude;
            return this;
        };
        this.setLongitude = function(longitude) {
            this.longitude          = longitude;
            this.lon                = longitude;
            this.coords.longitude   = longitude;
            return this;
        };

    }; // end class definition

    // instance factory method / constructor
    this.Position.prototype.instance = function(params) {
        return new
            this.constructor()
                    .setLatitude(params.latitude)
                    .setLongitude(params.longitude)
        ;
    };

    return new this.Position();

}) // end Factory Definition

.controller('testCtrl', function($scope, PositionFactory) {
    $scope.position1 = PositionFactory.instance({latitude: 39, longitude: 42.3123});
    $scope.position2 = PositionFactory.instance({latitude: 39, longitude: 42.3333});
}) // end controller

;
```

edited Mar 30 at 15:57

answered Mar 30 at 15:26

James Earlywine
11 2

Why did you answered such an old question, with lots of answers, one of them already accepted? – lodo Mar 30 at 15:36

1 Well, because I came here looking for an answer, and while many of the answers were helpful, I ultimately had to experiment a bit to come up with some code that makes sense to me, and the way I think of factories, and the way I've used them in other frameworks. Also my reputation is so small, I thought maybe I could improve my reputation, since I think this code is pretty good, and maybe it will help someone like me, who doesn't see entirely what they're looking for in the answers above. :) – James Earlywine Mar 30 at 15:41

I know a lot of excellent answer but I have to share my experience of using

1. `service` for most cases of default
2. `factory` used to create the service that specific instance

```
// factory.js ////////////////////////////
(function() {
'use strict';
angular
    .module('myApp.services')
    .factory('xFactory', xFactoryImp);
xFactoryImp.$inject = ['$http'];

function xFactoryImp($http) {
    var fac = function (params) {
        this._params = params; // used for query params
    };

    fac.prototype.nextPage = function () {
        var url = "/_prc";

        $http.get(url, {params: this._params}).success(function(data){ ...
    }
    return fac;
}
})();

// service.js ////////////////////////
(function() {
'use strict';
angular
    .module('myApp.services')
    .service('xService', xServiceImp);
xServiceImp.$inject = ['$http'];
```

```
function xServiceImp($http) {
    this._params = {'model': 'account','mode': 'list'};

    this.nextPage = function () {
        var url = "/_prc";

        $http.get(url, {params: this._params}).success(function(data){ ...
    }
}
})();
```

and using:

```
controller: ['xFactory', 'xService', function(xFactory, xService){

        // books = new instance of xFactory for query 'book' model
        var books = new xFactory({'model': 'book', 'mode': 'list'});

        // accounts = new instance of xFactory for query 'accounts' model
        var accounts = new xFactory({'model': 'account', 'mode': 'list'});

        // accounts2 = accounts variable
        var accounts2 = xService;
...
```

answered Apr 7 at 1:25

nguyên
677 11 20

This answer address the topic/question

how Factory, Service and Constant — are just syntactic sugar on top of a provider recipe?

or

how factory ,servic and providers are simailar interanlly

basically what happens is

when you make a `factory()` it sets you function provided in second argument to provider's `$get` and return it(`provider(name, {$get:factoryFn })`), **all you get is provider but no property/method other then `$get` of that provider(means you cant configure this)**

source of factory

```
function factory(name, factoryFn, enforce) {
    return provider(name, {
      $get: enforce !== false ? enforceReturnValue(name, factoryFn) : factoryFn
    });
  }
```

when making a service() it return you providing a factory() with a function that injects the constructor (return the instance of the constructor you provided in your service) and returns it

source code of service

```
function service(name, constructor) {
    return factory(name, ['$injector', function($injector) {
      return $injector.instantiate(constructor);
    }]);
  }
```

So basically in both cases you eventually get a providers $get set to your function you provided , but you can give anything extra than $get as you can originally provide in provider() for config block

edited Apr 13 at 23:06

answered Apr 13 at 22:55

A.B
7,142 1 6 25

---

this thread was very useful to me here's a demo app that exposes through example the use of most angular concepts https://github.com/oviwankenobi/AngularJSTodoList#angularjstodolist

answered Apr 24 at 7:13

Ovi-Wan Kenobi
84 9

There are good answers already, but I just want to share this one.

First of all: **Provider** is the way/recipe to create a `service` (singleton object) that suppose to be injected by $injector (how AngulaJS goes about IoC pattern).

And **Value, Factory, Service and Constant** (4 ways) - the syntactic sugar over **Provider** way/recepie.

There is `Service vs Factory` part has been covered: https://www.youtube.com/watch?v=BLzNCkPn3ao

Service is all about `new` keyword actually which as we know does 4 things:

1. creates brand new object
2. links it to its `prototype` object
3. connects `context` to `this`
4. and returns `this`

And **Factory** is all about Factory Pattern - contains functions that return Objects like that Service.

1. ability to use other services (have dependencies)
2. service initialization
3. delayed/lazy initialization

And this simple/short video: covers also **Provider**: https://www.youtube.com/watch?v=HvTZbQ_hUZY (there you see can see how they go from factory to provider)

Provider recipe is used mostly in the app config, before the app has fully started/initialized.

edited Jun 23 at 14:04

answered Jun 23 at 0:04

ses
4,344 10 53 111

protected by Pankaj Parkar Jun 11 at 9:39

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?